Research Report 233

The New Implementation of Cassandra

Iain D Craig

RR233

This paper describes our new, EuLisp, implementation of our CASSANDRA multi-agent architecture. Starting with a basic collection of structures, we constructed a sequential, a concurrent and a fully distributed version of the architecture. Each version was tested using the word recognition problem of Rumelhart and McClelland (the same problem we used to test previous versions of the architecture). We report on each of these versions and briefly describe our approach and experiences.

Department of Computer Science
University of Warwick
Coventry CV4 7AL
United Kingdom

October 1992

The New Implementation of CASSANDRA

Iain D. Craig
Department of Computer Science
University of Warwick
Coventry CV4 7AL
UK EC

October 20, 1992

Abstract

This paper describes our new, EuLisp, implementation of our CASSANDRA multi-agent architecture. Starting with a basic collection of structures, we constructed a sequential, a concurrent and a fully distributed version of the architecture. Each version was tested using the word recognition problem of Rumelhart and McClelland (the same problem we used to test previous versions of the architecture). We report on each of these versions and briefly describe our approach and experiences.

1 Introduction

In this paper, we describe our most recent implementation work on our CASSANDRA architecture [4]. Until this year, we have had to concentrate our experimental work on sequential implementations of this distributed AI architecture. With the arrival of the EuLisp [14] system, we were able to construct new implementations of CASSANDRA. Since February this year, we have constructed three versions of the architecture:

1. A sequential version (similar to the ones we have built in the past).
2. A concurrent version (which executes the Level Managers of a system in parallel—depending upon the hardware being used, this can be simulated concurrency or shared memory MIMD parallelism).
3. A fully distributed version.

The fully distributed version was our goal from the start of the CASSANDRA project. The demonstration system we have constructed executes on four processors connected by a LAN (Ethernet).

In a sense, we have completed the basic work on the architecture and have now shown that:

1. The architecture is general (this was shown by our initial pair of implementations).

2. The architecture is suitable for concurrent execution.

3. The architecture is, indeed, fully capable of supporting distributed processing.

As explained in the next section, the CASSANDRA architecture has changed somewhat over the years since we first began work on it. The EuLisp implementations completed this year were all based on the formal specification we presented in [7] (they omit one feature—communications preconditions—which we will discuss below in section 4.4). The latest work represents a more complete implementation of CASSANDRA in terms of functionality than hitherto.

The purpose of this current report is to describe the work that we have done on "completing" the architecture. That is, we will describe our parallel and distributed implementations of the architecture. In the next section, we outline some of the history of the architecture: this introduces some of the other work that we have done on CASSANDRA, in particular its formal definition (we believe that we were the first to produce a detailed formal definition of an AI architecture—the SOAR group at CMU have recently produced an equivalent account of SOAR [11]). Once historical matters have been dealt with, we describe the problem that our new implementations solve. This is the word recognition problem tackled by Rumelhart and McClelland [10, 15] which we described in [4] as our second CASSANDRA system. Next, we describe the three versions of CASSANDRA that we implemented earlier this year and their inter-relationships. In the final section, we summarize our experiences and point to the future.

It is assumed that the reader is already familiar with CASSANDRA. Those who do not possess this prior knowledge should consult either [4] or, preferably, [7].

Acknowledgements The work undertaken this year on CASSANDRA could not have been completed without EuLisp. We would like to thank Julian Padgett and Pete Broadberry (both of the University of Bath) for their help and advice on EuLisp and for considerable help in fixing bugs relating to the PVM interface. At Warwick, we would like to thank Roy Simmons and Jeff Smith for their support and provision of resources.

The text of this paper is based upon a lecture we delivered in the Department of Computer Science, Rijksuniversiteit Limburg, Maastricht, The Netherlands, on September 24, 1992. We would like to thank Dr P. J. Braspenning and Prof. dr. H. J. van den Herik for inviting us to give the lecture.

2 A Short History of CASSANDRA

During the past seven years, we have been developing the CASSANDRA architecture [4]. The architecture was designed as a development of the blackboard architecture [9, 13, 12, 3]. The new architecture was intended as a more modular version of the blackboard architecture, but was also intended to be a *distributed* problem-solving architecture. That is, CASSANDRA was intended for use in solving problems that manifest either physical or logical distribution. As a distributed architecture, CASSANDRA was faced with a number of problems that were not usually considered in AI systems: in particular, the needs of concurrent processing and the interactions between local processing and communications were considered in depth.

Our initial implementation of CASSANDRA solved an air-traffic control problem. This problem suggested many features that were eventually included in the architecture: in particular, the need for high reliability and for rapid processing led to some of the structures that are described in [4]. In order to show the generality of the architecture, we implemented a version that solved a small (but famous) problem in Cognitive Science. The problem is the word recognition one chosen by Rumelhart and McClelland for their pioneering work on parallel distributed processing [10, 15]. Our implementation of the word recognition problem was successful and demonstrated that CASSANDRA has some measure of generality.

Since the work reported in [4], we have completed a formal specification (in Z [16]). The principal reason for producing a formal (i.e., mathematical) specification of CASSANDRA was that we wanted to provide as unambiguous as possible a definition of the architecture. As we noted in [4], a formal specification would bring a number of benefits: removal of the ambiguity we believed to exist in some parts of the informal description in [4] was important, but we also wanted to prove *a priori* properties of the architecture. The formal specification that appears in [7] does not contain proofs of properties for we had decided to concentrate on the purely definitional part of the activity.

The definition of the architecture that appears in [7] differs somewhat from that in [4]. In his comparative study of blackboard and blackboard-based systems, Velthuijsen [17] notes that some aspects of the original CASSANDRA definition are left unspecified. We were aware of this also and had decided that the version of the definition that appears in [7] would rectify some of these problems[1]. In addition to the changes that appear between the two books, we have also discussed a number of significant extensions to CASSANDRA [5] (as we will mention in the last section, our ideas have changed in even more radical ways during the last year).

Unfortunately, our attempts at implementation have been considerably slower than our work on definition and theoretical matters. The initial implementations of

[1] Velthuijsen based his analysis upon the original CASSANDRA publications. In the appendix to his thesis, he notes that the definition changed in the passage from the original work to the formal specification.

CASSANDRA were all purely sequential, and ran on a single processor system. The LISP dialects we used did not provide mechanisms for handling concurrency, nor was the problem of interfacing CASSANDRA to a communications network (typically Ethernet) easily solved. Our problems in this respect have been solved by the use of EuLisp [14], a modern LISP dialect that includes facilities for object-oriented programming, concurrency, and which contains a library for interfacing EuLisp to the PVM demon [2]. PVM is a network demon that is used to pass messages between machines connected by a LAN, WAN or combination of both. With the advent of EuLisp and its libraries, we were able to complete our implementation of CASSANDRA and (finally) to use it as a fully distributed system that executes on more than one host machine.

3 The Word Recognition Problem

In this section, we briefly outline the problem that we used as a test of the new implementations of CASSANDRA. We will deliberately be brief: for more detailed descriptions, the reader should consult either the source material ([10, 15]) or the relevant chapter in [4]. Once we have outlined the problem, we will describe the generic implementation in CASSANDRA.

3.1 Word recognition

The problem is, essentially, as follows:
Convert a low-level representation of a sequence of letters into a representation that is suitable for lexical decision.

In other words, the task is to convert a low-level representation of a sequence of letters into a form such that a decision can be made as to whether the input represents a word in some language. The language we use is English, and the low-level representation is in terms of sets of features. We assume that these features have been extracted by a visual system that views the letters in printed form. Greater or lesser amounts of noise can be added to the input representation in order to determine the behaviour of the system under change of stimulus quality.

The task involves collecting sets of features and hypothesizing constituent letters. Once letters have been hypothesized, they are assembled into a sequence and a decision is made as to whether the letter sequence constitutes an English word. The processes that comprise the task require knowledge of valid Roman letters (i.e., knowledge of the alphabet used for English) and knowledge of English words. Letters must be hypothesized even when they are severely degraded by noise, and words must be hypothesized from less than perfect letter hypotheses. A letter hypothesis must be formed under all circumstances—including the circumstance in which the system is unable to decide what the letter is (i.e., it must guess).

The output of the task is a decision as to whether the input features comprise

an English word (the implementation also prints the best hypothesis). Unlike the experiments performed by Rumelhart and McClelland [10, 15] and like our original implementation [4], we make no assumptions about how many letters are present in words (there is a limitation of four letters, but this is based upon the contents of our dictionay, and not upon any other factor).

3.2 Generic implementation

The problem divides knowledge and processing into three logically distinct parts:

1. Input processing (processing of feature sets).
2. Letter hypothesis generation.
3. Word hypothesis generation.

This division forms the basic structure of the "generic" design or implementation. We refer to this as being "generic" because it represents the main divisions of the system that are used in each of the versions described in the next section.

To implement a CASSANDRA system that performs the word recognition task, three Level Managers are defined. Each Level Manager performs exactly one of the tasks listed in the last paragraph (there is no duplication of function). We refer to the Level Managers by the following names:

- **Perceptual**: this Level Manager handles the feature sets.
- **Letter hypothesizer**: this Level Manager performs letter hypothesis generation.
- **Word hypothesizer**: this Level Manager performs word hypothesis generation.

Communications are arranged as follows. The Perceptual Level Manager accepts input (in the form of a set of sets of features) from some external source. It sends messages to the Letter hypothesizer. The messages sent from the Perceptual to Letter Level Managers represent individual sets of features: each feature set corresponds to a single letter (with greater or lesser noise). The Letter hypothesizer processes the feature sets sent by the Perceptual Level Manager; the Letter hypothesizer also sends its letter hypotheses to the Word hypothesizer. The Word hypothesizer sends output to the external environment, but can also send messages back to the Letter hypothesizer Level Manager: this enables it to obtain new letter hypotheses when it discovers that one or more letters have been incorrectly hypothesized. When a rehypothesize message is received by the Letter hypothesizer, the Level Manager attempts to generate a new hypothesis for the letter in question.

Communication between the Letter and Word hypothesizer Level Managers is a simple form of dialogue. The Letter hypothesizer sends letter hypotheses one at a

time to the Word hypothesizer which collects them into words (this actually involves a small representation change, but sequence information is present in the message sent by the Letter hypothesizer, so the Word hypothesizer need only—in effect—generate a string from an ordered collection of characters). The Word hypothesizer then attempts to hypothesize a word that can be formed from the letters it has received (this is done by consulting a dictionary). If there is an exact match, the Word hypothesizer tells the user and then issues a termination message to the Letter hypothesizer. If the Word hypothesizer cannot find a word that exactly matches, it determines which letters it believes are at fault and sends one message for each faulty letter to the Letter hypothesizer: these messages ask the Letter hypothesizer to generate a new hypothesis. It can be seen that there is scope for considerable interaction (in terms of the number of messages) between the Word and Letter hypothesizer Level Managers.

The local databases of the Letter and Perceptual Level Managers contain representations of hypothesized letters and feature sets, respectively. The Word hypothesizer Level Manager stores letter hypotheses *as well as* word hypotheses in its local database. This scheme has some simplifying advantages: for example, the Word hypothesizer is able to check that the same hypothesis has not been made twice for the same letter position. The scheme also allows the use of a relatively small set of message types.

There is no global (i.e., system-wide) criterion for termination, nor is there a global condition on the correctness of the solution. Instead, messages are exchanged in order to perform these functions. For the word recognition task, three basic message types were defined. These types correspond to the primitive actions that can be performed on a traditional blackboard. The message types are:

- New. A message of this type informs the receiver that the message's content is a new entry which is to be posted in its local database.

- Mod. A message of this type contains an entry identifier, an attribute name and a value. A Mod message tells the receiver to modify one of the entries in its local database by setting the named attribute to the value that is in the message; the entry to be modified is named in the message.

- Add. A message of this type also contains the name of an entry, the name of an attribute and a value. When it receives an Add message, a Level Manager updates the entry named by the message (which, as with Mod messages, must exist in its local database) by adding the named attribute to the entry; the value of the newly added attribute is the value supplied in the message.

These message types are identical to those described in [4]. The reason for adopting this limited and low-level protocol is that it is quite well suited to the task. In addition, it reflects the relationship between CASSANDRA and blackboard systems.

Velthuijsen [17] adopts a similar approach to communication in his BLONDIE-III distributed blackboard system[2]. We now believe that the approach is excessively low level and that each Level Manager must know too much about the internal representations used in other Level Managers. However, we believe that the present approach is satisfactory for a system which uses relatively similar representations in all Level Managers. In addition, we were quite careful with our messages: they only mention attributes which relate to the *status* of an hypothesis, and they also only ever mention the names of entries which have been made public (in the sense that they have appeared in previous messages). As will be seen in the next section, the distributed implementation augments the set of message types.

Each Level Manager has its own set of knowledge sources. The knowledge sources used by a Level Manager are entirely private to the Level Manager in which they reside. The Perceptual Level Manager has two knowledge sources: one to process inputs and one to send messages to the Letter hypothesizer. The Letter and Word hypothesizers contain more knowledge sources, but, again, some knowledge sources are concerned with communications and not directly with problem-solving activities.

As with the implementation described in [4], control is relatively simple in the new implementation. Basically, the agenda is held as an unordered list of KSARs and the most recently added KSAR is always executed. This entails that the agenda is maintained using a LIFO discipline which implements a depth-first search.

The Letter and Word hypothesizers each make use of a database in addition to the ones prescribed by the CASSANDRA architecture. The Letter hypothesizer has a database which it uses in mapping letter feature sets to letters. The Word hypothesizer has a dictionary (in the current implementations, as in the previously documented one, the dictionary contains words of four letters). The dictionary is used to determine closeness of match between letter sequences and words. Letters are hypothesized using a best-first search over the feature mappings. In a similar way, word hypotheses are also searched in a best-first fashion. Best-first search uses closeness of match as its heuristic.

4 EuLisp Implementations

In this section, we describe the three EuLisp implementations of the CASSANDRA system for the word recognition problem. As will be seen, each of the implementations is based on the generic organization we described in the last section. The three implementations are (in historical order):

1. A sequential version.
2. A concurrent version.
3. A fully distributed version.

[2]Indeed, he based the scheme on ours—Personal communication, Summer, 1992.

Because of their novelty, we will concentrate on the last two.

4.1 Sequential

The sequential implementation was undertaken because, basically, CASSANDRA is a collection of sequential agents. Each agent functions as a single process and is unable to spawn sub-processes. This entails that the EuLisp implementation could be tested first using the Level Manager code in a sequential program: this is exactly what we did.

Each Level Manager is implemented as a collection of TEAOS classes[3]. The classes represent the standard CASSANDRA Level Manager constructs. More specifically, there are classes for each of:

- The local database.
- The local control databases.
- Knowledge sources.
- KSARs.
- Local database events.

The interpretation of these objects follows that in [7]—that is, we implemented the latest published version of the architecture (but see 4.4 below).

In addition to the above classes, a class was defined for the Level Manager itself. This class organized all the components of a Level Manager and provided methods to perform operations on the Level Manager's components. We called this organizing class the *engine* because it contains all the structures and operations required to execute a Level Manager.

The engine class must be instantiated for each Level Manager in a CASSANDRA system. Once instantiated, each engine must be equipped with a set of knowledge sources and code to support its local controller. When these additional objects have been provided and loaded into a Level Manager, the Level Manager is ready to execute.

The structure that we have just described is common to all three versions of the system. What differs in each case is what in [4] we called the *envelope*.

In the sequential version, we instantiated three engines, one for each of the Level Managers described in the last section. Each Level Manager was equipped with the necessary knowledge sources, local controller functions, and, in two cases (Word and Letter hypothesizers), external databases.

Communication channels in the purely sequential version were provided by FIFO queues. Methods were defined to add and remove items from an instance of the FIFO

[3] Actually, they are *structures*, but the differences between structures and classes need not bother us here. We will use the term 'class' because of its greater familiarity.

class. The method to add a message to a FIFO was used in knowledge source actions (so it still obeyed the constraints imposed and justified in [4]). The method to read from a queue was called in an envelope routine that called the engine. When a message was read from a queue (the method required a parameter that names the particular queue to be read), it was interpreted according to its type (New, Mod or Add). For the sequential version, therefore, a simple envelope was defined.

The FIFO queues were arranged so that they provided the channel structure described above: queues were only shared between a pair of Level Managers and could not be assigned in the sense that the Level Manager at each end of a FIFO could not be changed at runtime.

Because this version was completely sequential, a way of modelling the concurrent (distributed) operation of the CASSANDRA system had to be found. The method we adopted for the 1992 EuLisp implementation was identical to that described in [4]: each Level Manager was run until it had exhausted its agenda of executable KSARs. Each time the agenda became empty, another Level Manager was executed. A fixed sequence of Level Manager execution was adopted. The Perceptual Level Manager was first executed and ran to completion: the output was a sequence of messages to the Letter hypothesizer which were held in the FIFO that connected these two Level Managers.

Next, the Letter hypothesizer was executed. It began by reading the messages from the FIFO that connected it to the Perceptual Level Manager. Once the messages had been read and interpreted, the knowledge sources dealing with letter hypothesis formation were then executed. This led to the creation of messages to be sent to the Word hypothesizer Level Manager (they were sent on a FIFO that was used only for communication from the Letter to the Word hypothesizer and in that direction only—another FIFO was used for communication in the reverse direction).

The Word hypothesizer began by reading incoming messages. It then hypothesized words. If a perfect match was detected (every letter in the input sequence matched a word in the dictionary in the correct order), the Word hypothesizer presented its findings to the user and then terminated; termination of the Word hypothesizer caused the termination of the entire system. If letters were deemed incorrect, messages would be posted in the FIFO that connected the Word to the Letter hypothesizer (a different queue from the one which passed messages in the other direction). When the Word hypothesizer exhausted its stack of eligible KSARs, control was passed back to the Letter hypothesizer. Exhaustion of the eligible KSARs by the Word hypothesizer was considered to be different from termination: termination required the setting of a flag which caused the code which ran the Level Managers to exit.

The sequential implementation of the system was very close to the implementation described in [7] and it simulated concurrent and distributed processing in a similar way. The primary use of the sequential version was to test the CASSANDRA implementation. We should note that the use of EuLisp classes made for a

very clean implementation: indeed, we found that the modularity imposed upon us by this organization helped considerably in the other two versions.

4.2 Concurrent

We decided to implement a parallel version of CASSANDRA as an intermediate step between the purely sequential version and the fully distributed one. We believed that such an intermediate step would further determine the validity of the architecture.

As part of the standard language, EuLisp provides a *thread* abstraction as well as integer semaphores [14]. This makes a parallel implementation of CASSANDRA relatively straightforward, particularly if Level Managers are still implemented as sequential programs (we have considered some ways of parallelizing Level Managers [6], but have not found these ideas particularly satisfactory). The EuLisp concurrency primitives can be used on shared memory systems with one or more processors. We did not exploit the option of using operating system processes to implement EuLisp threads (this would have required recompiling EuLisp). Furthermore, due to current lack of available hardware, we could only execute the concurrent version on a single processor system[4].

Each Level Manager in the concurrent version was implemented as a single thread. That is, each Level Manager was an independent, concurrently executing entity within the system. The concurrent version contained three threads: one for each Level Manager.

The concurrent version was based on the sequential implementation described above. A number of changes needed to be made to the sequential CASSANDRA implementation of the word recognition task. These changes related mainly to the definition of a shared queue type and the definition of a new envelope for the Level Managers. We describe these in order.

As in the sequential version, channels are represented as FIFO queues. The channels are uni-directional and point-to-point; they are configured only once before execution begins. The scheme was adopted for the reasons that it is simple and that it corresponds closely to the semantics of channels that we gave in [4]. In the concurrent version, each queue becomes a critical section: we therefore extended the original FIFO type by the addition of a semaphore. Because of the EuLisp process model, it was also necessary to make an explicit call to the scheduler (the EuLisp scheduler, i.e.) in order to cause the next thread to execute.

Once the shared FIFO type had been defined, one shared FIFO was declared for each inter-Level Manager channel. The channels were declared exactly in accordance with the CASSANDRA definitions ([4, 7]).

The envelope was then defined. This was similar to the one used for the sequential version. Knowledge source actions were allowed to put messages on shared

[4] A multi-processor system with shared memory is becoming available at the Warwick University Computer Services as this report is written.

queues, but were not allowed to read messages—the envelope performed that function. Messages were read from an input shared FIFO once every engine cycle. Thus, messages were read only at specific times. The rest of the cycle consisted of executing the Level Manager's engine.

The engine was adopted *without* modification from the sequential version.

When an engine had no more KSARs to execute, the input FIFOs were checked to see if they contained messages. If there were messages, they were read and interpreted (thus causing local database events and, hence, causing the basic loop to repeat); if there were no messages in the input queues, the Level Manager's envelope suspended its thread. It should also be noted that a Level Manager could also be suspended when a knowledge source action sent a message on a queue: this is because of the explicit call to the EuLisp scheduler when the semaphore guarding the FIFO was released.

In the concurrent version, no explicit scheduling of Level Managers was performed. Instead, the EuLisp scheduler performed this task in its entirety.

Termination in the concurrent version was somewhat more complicated than in the sequential one. The scheme we adopted was to use a shared variable for signalling termination: the shared variable was set by the Word hypothesizer when it had either found a perfect hypothesis or had run out of useful hypotheses.

We expected that the concurrent version of the system would be slower than the sequential one. This was contrary to our findings. The concurrent version appeared to run as fast as the previous one. What we *did* find was that memory utilization increased when we moved to a concurrent implementation. We found the process of converting the sequential implementation to a concurrent one relatively easy, requiring, as it did, the definition of a new type and a few minor changes to the code. We also noted that the number of reschedules occurring when the Letter and Word hypothesizers were active were not excessive.

4.3 Fully distributed

The ultimate goal of the new implementation was the construction of a fully distributed version of CASSANDRA. The final step was, therefore, to use the Level Manager classes that we had developed in order to implement a fully distributed version.

EuLisp provides a library for interfacing to the PVM (*Parallel Virtual Machine*) demon [2]. PVM is an architecture-independent mechanism for the support of distributed systems. A PVM demon is started on a master processor and is supplied with a configuration file. The demon starts processes on each of the machines named in the configuration file and creates communications channels between them. A PVM demon must be present on each of the named machines (this must be done prior to starting the system, but is a simple exercise). The facilities provided by PVM are called from EuLisp via a set of functions provided by a library. These functions allow a number of operations to be performed, including sending and receiving

messages.

The natural distribution of the word recognition system was that of placing each Level Manager on a separate processor. In addition, we included a fourth processor for handling information messages and which configured the system. Configuration consists, basically, of starting a EuLisp process on each machine. Once the processes have been started, it is necessary to inform the EuLisp process of the operations it is to perform: in our case, we had to inform each EuLisp process of the Level Manager it was to execute. This was handled by arranging for the configuration node to send a configuration message to each EuLisp process. The recipient of each configuration message then loaded the appropriate Level Manager engine code and the its knowledge sources. Once this had been done, the Level Manger was ready to execute.

We could have arranged matters so that we used only three processors. To do this, we would have arranged for the configuration code to exit normally and then to load the Level Manager assigned to the configuration node's processor. We decided not to do this and to keep a separate configuration node running throughout the execution of the system. The reason we did this was that we wanted to be able to display a variety of information on the console. This information was contained in messages sent by the three Level Managers in the system and was included so that we could show that the system was really working. This feature is useful not only for doubting visitors or colleagues, but also for demonstrations to the public. Since we do not use a fancy graphic display for this application, there would otherwise be very little for anyone to see. Furthermore, the messages sent to the configuration node are also useful in determining the correct operation of the system.

We needed, of course, to write a configuration node that would start the remote Level Managers and then receive and display messages from the remote Level Managers. In addition, we had to define a set of message types and declare them to the PVM reader (the reader converts from LISP to external representation and vice versa—this is needed for transmission over the net). We took the opportunity to extend the repertoire of messages within the system. In particular, we defined the following additional types:

- An Error message type. This communicated error signals between the nodes. The intention was that a Level Manager would take specific actions upon receipt of a message of this type.

- An Inform message type. The messages exchanged between Level Managers and the configuration node are of this type. When the configuration node is setting up the network of Level Managers, it sends Inform messages to each of the remote processors. These messages contain the information needed to start the required Level Manager.

- A Termination message. This type was used to cause a Level Manager to terminate its activities and exit its envelope. The Word hypothesizer sent

Termination messages to the Letter hypothesizer to indicate (i) that it was terminating, and (ii) to force the Letter hypothesizer to terminate.

We were lucky that the Error message type turned out not to be necessary. However, the Inform type formed the basis of what passes for the user interface. The Termination message type was used in the way explained above and implemented the basic distributed termination mechanism.

It should be noted that we configured the Perceptual Level Manager so that it could only ever accept one input (which was stored in a variable local to the Perceptual Level Manager's process—it could have been in a file or could have been provided interactively). Once the Perceptual Level Manager had passed its messages to the Letter hypothesizer, the Perceptual Level Manager terminated, but did not bother to inform either of the other Level Managers (it will be recalled that they do not need to interact with it once it has performed its task)—it did, though, send an Inform message to the configuration node so that the user could be informed of its termination. A variation to the system is to allow the Perceptual Level Manager to process more than one collection of input features: this would be relatively simple to implement.

Initially, we expected that we would have to buffering threads in the envelopes of the three Level Managers to buffer incoming and outgoing messages. It turned out that this was not necessary: the PVM primitives were such that we did not encounter any major performance problems, nor did we find that reception delays were important.

The envelope that we used was similar to that used in the sequential and concurrent versions: the main difference was that calls to PVM primitives replaced calls to FIFO queue methods. Message transmission could occur at any time, but reception could only occur at a single point in the envelope's loop. When a message was received, the engine was executed and continued to execute until there were no more executable KSARs. We could have adoped a scheme in which it was possible to receive incoming messages more than once per cycle, but this would have entailed a significant change to the engine code. We do note, however, that the arrangement we adopted for the concurrent and fully distributed versions of CASSANDRA may be inadequate when there is a large volume of messages needing to be processed.

For the fully distributed version, we added an extra knowledge source to the Word hypothesizer. This additional knowledge source monitored the quality of the current best word hypothesis. When the solution quality degraded with respect to previous best hypotheses, the knowledge source terminated the entire system. This knowledge source was introduced as a better check on solution quality. With the exception of obvious changes to knowledge sources caused by a change in communications medium, the initial complement of knowledge sources in this version was the same as in the other two.

The fully distributed version is exactly that. It does not use any shared variables, and each Level Manager executes in a completely separate address space on

a completely separate processor. The processors could be connected to a LAN or they could be connected to a WAN: these differences are immaterial to PVM and to the CASSANDRA code.

4.4 Omissions

Before ending this section, we must note one omission. We did not implement the communications preconditions described in [4] and formally defined in [7]. When we began the EuLisp implementation, we intended to include communications preconditions. For the word recognition task, we were already aware that we had no real need for this additional knowledge source slot, particularly if we adopted the same approach to communications that we had used in [4]. Nevertheless, we did examine in detail how we would implement them. We found that the complete implementation of communications preconditions was highly complex, and, we believe, would entail a significant overhead at runtime (because of the need to route messages within a Level Manager and because of the need to locate destination KSARs in the case in which a message does not name a specific KSAR). Furthermore, completely general communications preconditions can be satisfied by sets of messages: such a precondition can receive a number of messages, perhaps requiring them to be presented in some particular order. Completely general communications preconditions also impose a considerable overhead in terms of space because partial state must be retained. We concluded that the construct was probably not as well-defined as we had thought (even though we had already given a formal specification) and so decided to omit it from the current implementation.

5 Conclusions

In this paper, we have described our efforts to implement a new version of our CASSANDRA architecture using the EuLisp programming language. This effort resulted in three versions: a sequential, a concurrent and a properly distributed one. The concurrent and distributed versions of the system were constructed from components developed for the sequential version: because each CASSANDRA Level Manager is a single sequential process, this is a viable implementation approach. We have tested the implementations on an example problem: the word recognition problem first described in [10, 15]. In each case, we were pleased to discover that CASSANDRA functioned as well as we had hoped.

The new implementation omits the communications preconditions we introduced in [4] and formally defined in [7]. We found that the concept in its full generality was too complex to include before more work on the concept had been done.

We also note that the communications mechanisms used in this version of CASSANDRA are relatively poor. They resemble the kinds of communication used by conventional distributed systems. We presented a very brief analysis of this aspect

of the system above, but believe that more attention needs to be paid to communications. We argued in [8] that there are significant problems with the approach adopted for communications in CASSANDRA.

We have now completed the development of CASSANDRA. This work began in 1985 and was, as we noted above, hampered by a lack of facilities for distributed processing. With the availability of EuLisp, we were able to move from single processor, sequential implementations to fully distributed ones. During the seven years that it has taken to get this far, we have proposed a number of extensions to CASSANDRA (see [5] for details), but, more recently, we have begun to see some problems with the general approach we adopted in 1985 (see [8]). We are now exploring these problems and are in the course of defining a new architecture to replace CASSANDRA. Meanwhile, CASSANDRA is not defunct, and we hope others will continue to use it.

References

[1] Baum, L. S., Dodhiawala, R. T. and Jagannathan, V., The Erasmus System, *Proc. AAAI Workshop on Blackboard Systems Implementation Issues*, Seattle, WA, 1987.

[2] Beguelin, A., Dongarra, J., Geist, A, Manchek, R. and Sunderam, V., *A User's Guide to PVM Parallel Virtual Machine*, Technical Report ORNL/TM-11826, Engineering Physics and Mathematics Division, Mathematical Sciences Section, Oak Ridge National Laboratory, Oak Ridge, Tenessee, 1992.

[3] Craig, I. D., Blackboard Systems, *Artificial Intelligence Review*, Vol. 2, pp. 103-118, 1988.

[4] Craig, I. D., *The CASSANDRA Architecture*, Ellis Horwood, Chichester, UK, 1989.

[5] Craig, I. D., *Extending CASSANDRA*, Research Report No. 183, Department of Computer Science, University of Warwick, 1991.

[6] Craig, I. D., *Making CASSANDRA Parallel and Distributed*, Research Report No. 180, Department of Computer Science, University of Warwick, 1991.

[7] Craig, I. D., *The Formal Specification of Advanced AI Architectures*, Ellis Horwood, Chichester, UK, 1991.

[8] Craig, I. D., *Replacing CASSANDRA*, Research Report No., Department of Computer Science, University of Warwick, 1992 (in prep.).

[9] Hayes-Roth, B., A Blackboard Model for Control, *Artificial Intelligence*, Vol. 26, pp. 251-322, 1985.

[10] McClelland, J. L. and Rumelhart, D. E., An interactive model of context effect in letter perception: Part 1. An account of basic findings, *Psychological Review*, Vol. 88, pp. 375-401, 1981.

[11] Milnes, B. et al., *A Specification of the Soar Cognitive Architecture in Z*, School of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, 1992.

[12] Morgan, T. and Engelmore, R., *Blackboard Systems*, Addison-Wesley, Wokingham, UK, 1988.

[13] Nii, H. P., The Blackboard Model of Problem Solving, *AI Magazine*, Vol. 7, pp. 38-53, 1986.

[14] Padgett, J. A. et al., *EuLisp Definition Version 0.7.5*, School of Mathematics, University of Bath, 1992.

[15] Rumelhart, D. E. and McClelland, J. L., An interactive model of context effect in letter perception: Part 2. The contextual enhancement effect and some tests and extensions of the model, *Psychological Review*, Vol. 89, pp. 60-94, 1982.

[16] Spivey, J. M., *The Z Notation*, Prentice Hall, Hemel Hempstead, UK, 1989.

[17] Velthuijsen, H., *The Nature and Applicability of Blackboard Systems*, Ph. D. Thesis, Department of Computer Science, Limburg University, Maastricht, The Netherlands, 1992.